L3-MI / PROG
TP#7

Pierre Karpman

2024-11-06

Quicksort

Le but de cet exercice est d'implémenter l'algorithme de tri *quicksort*, d'abord sur un tableau d'entiers puis pour des structures de donnée arbitraires dotées d'une fonction de comparaison.

On commence par rappeler l'algorithme de *partition en place* en ***pseudocode***, l'algorithme de tri rapide complet pouvant s'obtenir par une série d'appels récursifs à ce dernier. Les premières questions sont pour une partition utilisant le dernier élément comme *pivot* (c'est donc un algorithme déterministe), et les questions suivantes sont pour un pivot aléatoire (le « vrai » tri rapide).

```
/****************************************
PARTITION
input: T
input: d, f, x = T[f]
output: n, la nouvelle position de x
dans le tableau modifié en place t.q.
T[d..n-1] <= x; T[n+1..f] > x
****************************************/
x = T[f];
n = f;
for (i = f - 1; i >= d; i--)
{
        if (T[i] > x)
        {
                T[n] = T[i];
                T[i] = T[n-1];
                n = n - 1;
        }
}
T[n] = x;
```

Remarque : Lors de l'implémentation de cet algorithme, faites particulièrement attention aux bornes sur `T` et vos variables de boucles ; une utilisation préventive d'ASan et de Valgrind permet de détecter de nombreux bugs classiques.

Q.1 : Écrivez une fonction `bool sorted(size_t Tlen, const uint32_t T[Tlen])` qui teste si un tableau `T` de `Tlen` `uint32_t` est trié.

Q.2 :

1. Écrivez une fonction `void qsortu(size_t Tlen, uint32_t T[Tlen])` qui trie en place un tableau `T` de `Tlen` `uint32_t` en utilisant quicksort.
2. Testez votre fonction. Ces tests devront au minimum utiliser `sorted`, sur des tableaux remplis aléatoirement. Pourquoi l'utilisation de `sorted` ne permet elle pas de détecter tous les bugs possibles dans `qsortu` ?
3. Évaluez expérimentalement le coût de votre implémentation en mesurant le temps d'exécution sur des tableaux de taille variable (par ex. de quelques millions à quelques centaines de millions d'éléments remplis aléatoirement. Faites attention à tirer vos éléments aléatoires (par ex. uniformément) dans un ensemble suffisamment grand par rapport à la taille du tableau (pourquoi ?), et à exclure le temps de génération des tableaux de vos mesures. Vous pourrez par exemple utiliser la fonction `clock_gettime` pour une mesure raisonnablement précise du temps d'exécution.

Q.3 :

1. Que pouvez-vous prévoir sur le temps d'exécution de votre tri sur un tableau dont les éléments sont initialement strictement décroissants ? Vérifiez le expérimentalement sur de petits tableaux de tailles variables.
2. Modifiez votre fonction de partition afin de résoudre ce problème en moyenne. Une solution suffisante est d'échanger `T[f]` avec un élément au hasard (uniforme) de `T[d..f]`, puis de procéder comme auparavant.
3. Testez l'effet de votre changement sur le cas problématique précédent.

Q.4 :

1. Écrivez une nouvelle fonction `qsortg` basée sur votre fonction `qsortu` pour qu'elle offre le même prototype et comportement que la fonction `qsort` de la bibliothèque C standard. Vous pourrez par exemple utiliser `memcpy` pour effectuer les affectations nécessaires.

   Remarque : La logique de `qsortg` est identique à `qsortu` (et de même pour leurs fonctions de partition respectives) ; il y a donc peu de changements à faire à `qsortu` pour obtenir `qsortg`.
2. Comparez les performances de vos deux implémentations (`qsortu` et `qsortg`, toutes deux avec un choix aléatoire du pivot) ainsi que celle de la bibliothèque standard pour des tableaux d'`uint32_t`. D'où peuvent selon vous provenir d'éventuelles différences ?